Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE- SEMESTER–III (NEW) EXAMINATION – WINTER 2024

Subject Code: 3130703 Date: 29-11-2024

Subject Name: Database Management Systems

Time: 10:30 AM TO 01:00 PM Total Marks: 70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			MARKS
Q.1	**(a)**	Define following terms. i) Database Management System ii) Instance iii) Logical Data independence	**03**
	(b)	List and explain categories of database users. Describe roles and responsibilities of database administrator.	**04**
	(c)	Compare the advantages of using a Database Management System (DBMS) over a file processing system.	**07**
Q.2	**(a)**	Explain generalization and specialization with neat diagram	**03**
	(b)	Explain following attributes. a. Single Valued b. Multivalued c. Derived attribute d. Composite attribute	**04**
	(c)	Draw an E-R diagram of following scenario. Make necessary assumptions and clearly note down the assumptions. Municipal Corporation/any Bus reservation system should be digitize.	**07**
		OR	
	(c)	Explain the concepts of strong entity and weak entity using real world example.	**07**
Q.3	**(a)**	Explain the terms. i) Super Key ii) Foreign Key iii) Unique Key	**03**
	(b)	Explain trivial functional dependencies with suitable example.	**04**
	(c)	Consider a relation R(A, B, C, D, E) with the following three functional dependencies. AB → C ; BC → D ; C → E; Find out the number of superkeys in the relation R.	**07**
		OR	
Q.3	**(a)**	Explain insertion and deleting anomalies with respect to normalization.	**03**
	(b)	Explain Armstrong's axioms in detail.	**04**
	(c)	Given a relation R(P, Q, R, S, T) and Functional Dependency set FD = { PQ → R, S → T }, determine whether the given R is in 2NF? If not convert it into 2 NF.	**07**

Q.4	**(a)**	Illustrate various storage strategies.	**03**
	(b)	Explain authorization and authentication with respect to database security.	**04**
	(c)	Which kind of queries are solved using division operator? Explain in detail.	**07**

OR

Q.4	**(a)**	Differentiate dynamic hashing and static hashing	**03**
	(b)	Explain ACID properties of transaction.	**04**
	(c)	Describe query processing with neat diagram.	**07**

Q.5	**(a)**	Explain working of two phase locking protocol.	**03**
	(b)	Explain GRANT, REVOKE and SAVEPOINT commands with suitable example.	**04**
	(c)	Assume table CUSTOMER (Cust_Id,Customer_name, Age,Address,Salary). Write a PL/SQL function which givens total number of customers having salary more than one lac per month.	**07**

OR

Q.5	**(a)**	Differentiate between conflict and view serializability with respect to transaction.	**03**
	(b)	Categorize joins in the SQL. Explain each with suitable example.	**04**
	(c)	Write a PL/SQL trigger where employee of "GTU Private Ltd." company cannot update database on 23-Dec-2024 due to maintenance.	**07**

****

Q.1 (a) Define following terms. i) Database Management System ii) Instance iii) Logical Data independence (3 mark)

i) Database Management System (DBMS):
A **Database Management System (DBMS)** is a software system that is used to **create, store, manage, and retrieve data** from a database in an efficient and organized manner. It acts as an **interface between users/applications and the database**, ensuring that data is accessed in a controlled way. A DBMS provides facilities for **data definition, data manipulation, data security, concurrency control, and backup and recovery**. Examples of DBMS include Oracle, MySQL, SQL Server, and PostgreSQL.

ii) Instance:
An **instance** of a database refers to the **actual data stored in the database at a particular moment of time**. It represents the **current state or snapshot** of the database. As data is inserted, deleted, or updated, the instance of the database changes. Unlike the database schema, which is relatively stable, the instance is **dynamic** and keeps changing with database operations.

iii) Logical Data Independence:
Logical data independence is the ability to **change the logical schema** of a database without affecting the **external schema or application programs**. It allows modifications such as adding or removing attributes, tables, or relationships without requiring changes in user views or application code. Logical data independence improves **flexibility, maintainability, and scalability** of the database system.

(b) List and explain categories of database users. Describe roles and responsibilities of database administrator. (4 mark)

Introduction:
In a database system, different users interact with the database in different ways depending on their **requirements and responsibilities**.

These users are broadly classified into various categories. Among them, the **Database Administrator (DBA)** plays a crucial role in managing and controlling the entire database system to ensure **security, performance, and reliability**.

Categories of Database Users:

1. **Naive (Parametric) Users:**

- These users interact with the database using **predefined application programs**.
- They do not need knowledge of SQL or database structure.
- Example: Bank tellers, data entry operators, reservation clerks.

2. **Application Programmers:**

- These users develop **application software** that accesses the database.
- They write programs using programming languages such as **Java, C++, or Python** along with embedded SQL.
- They design forms, reports, and business logic.

3. **Sophisticated Users:**

- These users interact directly with the database using **query languages like SQL**.
- They understand the database structure and perform complex queries.
- Example: Data analysts, engineers, researchers.

4. **Specialized Users:**

- These users write **special-purpose database applications** for complex tasks.

- Applications may involve **scientific data, multimedia data, or artificial intelligence**.
- Example: CAD engineers, AI system developers.

Role and Responsibilities of Database Administrator (DBA):

- **Database Design:** Defines the **database schema, tables, relationships, and constraints**.
- **Security Management:** Controls **user authorization, authentication, and access rights**.
- **Performance Monitoring:** Monitors database performance and **optimizes queries and storage**.
- **Backup and Recovery:** Ensures **regular backups** and recovers data in case of system failure.
- **Concurrency Control:** Manages **simultaneous access** to the database to maintain consistency.
- **Maintenance:** Performs routine tasks such as **upgrades, tuning, and integrity checks**.

(c) Compare the advantages of using a Database Management System (DBMS) over a file processing system. (7 mark)

Introduction:
A **file processing system** stores data in separate files managed by application programs, whereas a **Database Management System (DBMS)** stores data in an integrated and organized manner under the control of a centralized system. Traditional file systems suffer from many limitations such as data redundancy, inconsistency, and poor security. A DBMS overcomes these drawbacks and provides several advantages, making it suitable for modern data-intensive applications.

Advantages of DBMS over File Processing System:

1. **Reduced Data Redundancy:**

- In a file processing system, the same data is stored in multiple files, leading to **duplication of data**.
- DBMS stores data in a **centralized database**, minimizing unnecessary duplication and saving storage space.

2. **Improved Data Consistency:**

- Redundant data in file systems often causes **inconsistencies** when updates are not applied everywhere.
- DBMS ensures **consistent data** because updates are made at a single location.

3. **Better Data Security:**

- File systems provide **limited security** mechanisms.
- DBMS offers **strong security features** such as user authentication, authorization, access control, and encryption.

4. **Data Integrity:**

- File systems lack proper mechanisms to enforce data validity.
- DBMS enforces **integrity constraints** like primary key, foreign key, and domain constraints to maintain correctness of data.

5. **Data Sharing and Concurrency Control:**

- In file systems, simultaneous access may lead to **data corruption**.
- DBMS supports **concurrent access** using locking and transaction management while preserving consistency.

6. **Backup and Recovery:**

- File systems require **manual backup and recovery**, which is error-prone.
- DBMS provides **automatic backup and recovery mechanisms** to protect data from system failures.

7. **Data Independence:**

- File systems are highly dependent on application programs.
- DBMS provides **logical and physical data independence**, allowing changes without affecting applications.

8. **Efficient Data Access:**

- Searching data in file systems is often **slow and inefficient**.
- DBMS uses **indexes and query optimization** techniques for faster data retrieval.

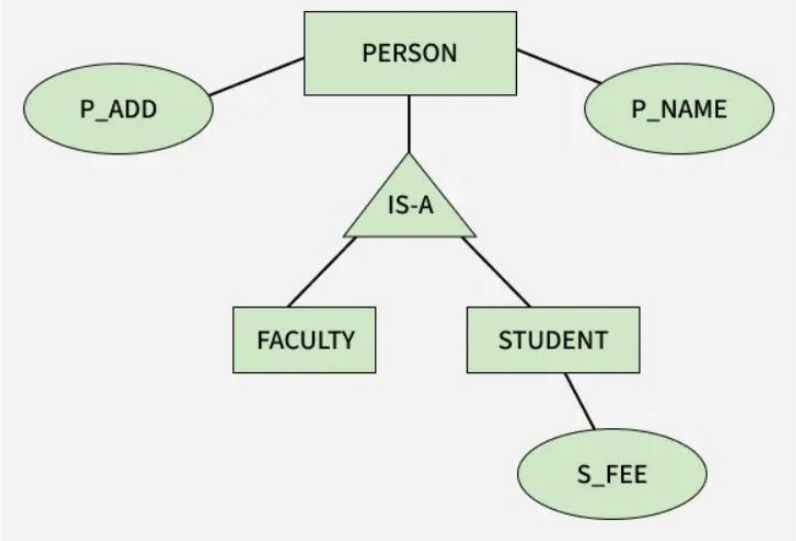
9. **Support for Transactions:**

- File systems do not support **ACID properties**.
- DBMS ensures **Atomicity, Consistency, Isolation, and Durability** for reliable transaction processing.

Q.2 (a) Explain generalization and specialization with neat diagram. (3 mark)

Generalization

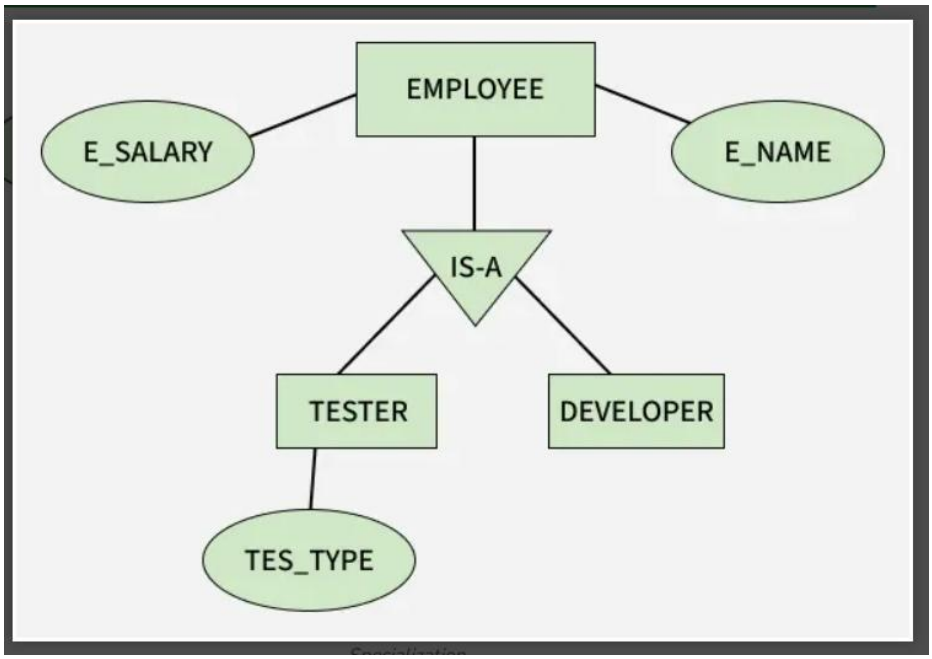
Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher-level entity if they have some attributes in common.



Example: *STUDENT and FACULTY can be generalized to a higher-level entity called PERSON as shown in diagram below. In this case, common attributes like P_NAME and P_ADD become part of a higher entity (PERSON) and specialized attributes like S_FEE become part of a specialized entity (STUDENT).*

Specialization

In specialization, an entity is divided into sub-entities based on its characteristics. It is a top-down approach where the higher-level entity is specialized into two or more lower-level entities



Example: *an EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER, etc. In this case, common attributes like E_NAME, E_SAL, etc. become part of a higher entity (EMPLOYEE) and specialized attributes like TES_TYPE become part of a specialized entity (TESTER).*

(b) Explain following attributes. a. Single Valued b. Multivalued c. Derived attribute d. Composite attribute (4 mark)

Introduction:

In the **Entity–Relationship (ER) model**, attributes are the **properties or characteristics** that describe an entity. Based on their nature and values, attributes are classified into different types. Understanding these attribute types helps in **accurate database design** and proper representation of real-world data.

---

a) Single-Valued Attribute:

- A **single-valued attribute** holds **only one value** for each entity instance.
- It represents a **simple and atomic property** of an entity.
- Example:
  - Roll_No of a student
  - Age of a person
- For each entity, the value of this attribute **cannot have multiple entries**.

---

b) Multivalued Attribute:

- A **multivalued attribute** can hold **more than one value** for a single entity.

- It is used when an entity can have **multiple occurrences of the same attribute**.
- Example:
  - Phone_Number of a student
  - Skills of an employee
- In ER diagrams, multivalued attributes are usually shown using a **double oval**.

---

c) Derived Attribute:

- A **derived attribute** is one whose value is **calculated from other attributes**.
- It is **not stored directly** in the database but computed when required.
- Example:
  - Age derived from Date_of_Birth
  - Total_Marks derived from individual subject marks
- Derived attributes help in **reducing data redundancy**.

---

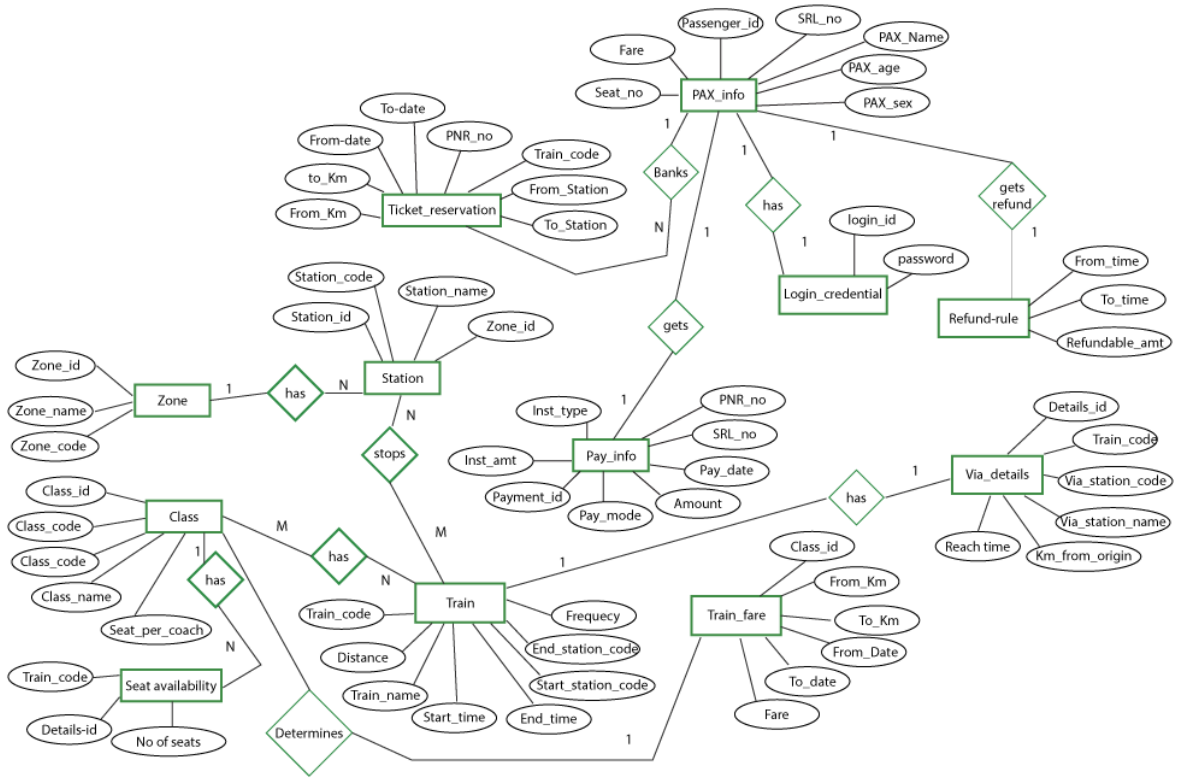
d) Composite Attribute:

- A **composite attribute** can be **divided into smaller sub-attributes**.
- It represents a complex attribute made up of meaningful components.
- Example:
  - Name → First_Name, Middle_Name, Last_Name

- Address → Street, City, State, PIN

- Composite attributes improve **data organization and clarity**.

(c) Draw an E-R diagram of following scenario. Make necessary assumptions and clearly note down the assumptions. Municipal Corporation/any Bus reservation system should be digitize. (7 mark)



OR

(c) Explain the concepts of strong entity and weak entity using real world example. (7 mark)

Introduction / Definition:

In the **Entity–Relationship (ER) model**, entities are classified into **strong entities** and **weak entities** based on their ability to be uniquely identified. This classification is important for designing databases that correctly represent **real-world relationships** and maintain **data integrity**.

Strong Entity:

Definition:
A **strong entity** is an entity that has its **own primary key**, which uniquely identifies each record in the entity set. It does **not depend on any other entity** for its existence.

Characteristics:

- Has a **primary key** of its own.
- Exists **independently** in the database.
- Does not rely on another entity for identification.
- Represented by a **single rectangle** in an ER diagram.

Real-World Example:
Student Entity

- Attributes: Student_ID (PK), Name, DOB, Gender, Branch
- Each student can be uniquely identified using Student_ID.
- Even if related entities are removed, the student entity can still exist.

Explanation (in words):
In a college database, each student is identified by a unique roll number. This roll number acts as a primary key, making **Student a strong entity**.

Weak Entity:

Definition:
A **weak entity** is an entity that **does not have a primary key of its own**. It depends on a **strong (owner) entity** for its identification and existence.

Characteristics:

- Does **not have a primary key**.
- Identified using a **partial key** along with the primary key of the strong entity.
- **Existence-dependent** on a strong entity.
- Represented by a **double rectangle** in an ER diagram.

Real-World Example: Dependent Entity (Employee–Dependent relationship)

- Strong Entity: Employee (Emp_ID)
- Weak Entity: Dependent (Dep_Name, Age, Relation)

Explanation (in words):
A dependent cannot exist without an employee. To identify a dependent, we need both:

- Emp_ID (from Employee)
- Dep_Name (partial key)
  Together, they uniquely identify the dependent. Hence, **Dependent is a weak entity**.

Identifying Relationship:

- A weak entity is connected to its strong entity using an **identifying relationship**.
- This relationship ensures that the weak entity **cannot exist independently**.
- In ER diagrams, this relationship is shown using a **double diamond**.

Comparison Summary:

Aspect	Strong Entity	Weak Entity
Primary Key	Has its own	Does not have
Dependency	Independent	Dependent on strong entity
Identification	Self-identified	Uses partial key + owner key
Existence	Independent	Cannot exist alone

Applications:

- Used in **banking systems** (Account–Transaction)
- Used in **employee management systems** (Employee–Dependent)
- Used in **order systems** (Order–Order_Items)

Q.3 (a) Explain the terms. i) Super Key ii) Foreign Key iii) Unique Key (3 mark)

Introduction:
In a relational database, **keys** are used to **identify records uniquely** and to establish **relationships between tables**. Keys also help in maintaining **data integrity and consistency** within the database. The most commonly used keys are super key, foreign key, and unique key.

---

i) Super Key:

A **super key** is a set of **one or more attributes** that can uniquely identify a tuple (row) in a relation. It may contain **extra attributes** that are not necessary for unique identification. Every table can have **multiple super keys**, but not all super keys are minimal.

- Example: In a Student table, {Roll_No}, {Roll_No, Name} are super keys because they uniquely identify a student.

---

ii) Foreign Key:

A **foreign key** is an attribute or a set of attributes in one table that **refers to the primary key of another table**. It is used to **establish and maintain relationships** between two tables and ensures **referential integrity**.

- Example: StudID in Issue_Books table is a foreign key referencing StudID in Students table.

---

iii) Unique Key:

A **unique key** is an attribute or a set of attributes that ensures **all values are unique** across the table. It prevents duplicate entries but can allow **NULL values** (depending on DBMS). A table can have **multiple unique keys**, unlike a primary key.

- Example: Email_ID in a user table can be defined as a unique key.

(b) Explain trivial functional dependencies with suitable example. (4 mark)

Introduction / Definition:
In relational database design, a **functional dependency (FD)** represents a relationship between two sets of attributes in a relation, where the value of one attribute (or set of attributes) **determines** the value of another attribute. A **trivial functional dependency** is a special type of functional dependency that always holds true by definition and plays an important role in **normalization and inference rules**.

Definition of Trivial Functional Dependency:
A functional dependency **X → Y** is said to be **trivial** if **Y is a subset of X**, that is, **Y ⊆ X**.

- This means the attributes on the right-hand side are already included in the left-hand side.
- Trivial functional dependencies are **always satisfied**, regardless of the data stored in the relation.

Explanation:

- Since the dependent attributes are part of the determining attributes, there is **no new information** being derived.
- Trivial FDs do not impose any additional constraint on the database.
- They are mainly used in **theoretical concepts**, such as **Armstrong's axioms** and **normalization processes**.

Examples:
Consider a relation:
STUDENT (Roll_No, Name, Branch)

Some trivial functional dependencies are:

- {Roll_No, Name} → Name
- {Roll_No, Branch} → Roll_No
- {Roll_No, Name, Branch} → Name

In all the above cases, the right-hand side attribute already exists in the left-hand side, so these dependencies are trivial.

Key Points / Characteristics:

- Always **true by definition**.
- Do not provide **new information**.
- Useful in **deriving other functional dependencies**.
- Based on the **reflexivity rule** of Armstrong's axioms.

Advantages / Importance:

- Helps in understanding **functional dependency theory**.
- Simplifies the process of **dependency inference**.
- Assists in **normal form analysis** during database design.

(c) Consider a relation R(A, B, C, D, E) with the following three functional dependencies. AB → C ; BC → D ; C → E; Find out the number of superkeys in the relation R. (7 mark)

Introduction:

In relational database theory, **keys and superkeys** play a vital role in uniquely identifying tuples in a relation. A **superkey** is any set of attributes that can uniquely identify a tuple, while a **candidate key** is a **minimal superkey**. To find the number of superkeys, we must first identify the **candidate key(s)** of the given relation using **attribute closure**.

Step 1: Given Relation and Functional Dependencies

Relation:
R(A, B, C, D, E)

Functional Dependencies (FDs):

- AB → C
- BC → D
- C → E

Step 2: Find Candidate Key using Attribute Closure

We calculate the **closure of AB**, since it appears on the left side of a dependency.

Closure of AB (AB^+):

- Start with: AB
- From **AB → C**, add C → ABC
- From **C → E**, add E → ABCE
- From **BC → D**, since B and C are present, add D → **ABCDE**

Since $\mathbf{AB^+ = \{A, B, C, D, E\}}$, AB can determine all attributes of the relation.

✓ Therefore, **AB is a candidate key**.

Step 3: Check for Other Candidate Keys

- $A^+ = A$ (not a key)
- $B^+ = B$ (not a key)
- $C^+ = CE$ (not a key)
- $BC^+ = BCDE$ (missing A, not a key)

✓ Hence, **AB is the only candidate key**.

Step 4: Determine Superkeys

Definition:
A **superkey** is any set of attributes that **contains a candidate key**.

- Candidate Key = **AB**
- Remaining attributes = **{C, D, E}**

Any combination of these remaining attributes added to AB will form a **superkey**.

Step 5: Calculate Number of Superkeys

Number of remaining attributes = 3
Number of subsets of {C, D, E} = $\mathbf{2^3 = 8}$

So, the **total number of superkeys = 8**

List of Superkeys:

1. AB
2. ABC
3. ABD
4. ABE
5. ABCD
6. ABCE
7. ABDE
8. ABCDE

OR

Q.3 (a) Explain insertion and deleting anomalies with respect to normalization. (3 mark)

Introduction / Definition:
In database systems, **anomalies** are problems that occur in a database due to **poor table design and data redundancy**. These anomalies mainly arise when a relation is **not properly normalized**. **Insertion anomaly** and **deletion anomaly** are two important types of anomalies that highlight the need for **normalization** in relational database design.

Insertion Anomaly:

- An **insertion anomaly** occurs when **new data cannot be inserted** into the database without inserting some **unwanted or unrelated data**.
- This happens because multiple facts are stored in a **single table**.
- Example (in words):
    - Consider a table storing **Student_ID, Student_Name, Course, and Faculty**.
    - If a new course is introduced but no student has enrolled yet, the course **cannot be inserted** without adding dummy student data.
- This leads to **data inconsistency and redundancy**.

Deletion Anomaly:

- A **deletion anomaly** occurs when **deleting a record unintentionally removes important information** from the database.
- This also results from storing **multiple facts together** in one relation.
- Example (in words):
    - If the only student enrolled in a course is deleted, the information about the **course itself is also lost**.
- This causes **loss of valuable data**.

Role of Normalization:

- **Normalization** divides large tables into smaller, well-structured relations.
- It helps in **removing data redundancy** and **eliminating anomalies**.

- Proper normalization ensures **safe insertion, update, and deletion operations**.

(b) Explain Armstrong's axioms in detail. (4 mark)

Introduction / Definition:
Armstrong's axioms are a set of **inference rules** used to derive all possible **functional dependencies (FDs)** from a given set of functional dependencies in a relational database. These axioms form the **foundation of normalization theory** and help in determining keys, checking dependency implication, and designing well-structured databases.

Armstrong's Axioms:

Armstrong proposed **three basic axioms** which are **sound and complete**, meaning all valid functional dependencies can be derived using them.

---

1. Reflexivity Rule:

- If **Y is a subset of X**, then **X → Y** holds true.
- This rule represents **trivial functional dependencies**.

Example:

- If X = {A, B, C}, then
  - {A, B, C} → A
  - {A, B} → B

Explanation:
Since the dependent attribute already exists in the determinant, the dependency is always true.

---

2. Augmentation Rule:

- If **X → Y** holds, then **XZ → YZ** also holds for any attribute set Z.

Example:

- If A → B is true, then
    - AC → BC is also true

Explanation:
Adding the same attributes to both sides of a functional dependency does not change its validity.

---

3. Transitivity Rule:

- If **X → Y** and **Y → Z** hold, then **X → Z** also holds.

Example:

- If A → B and B → C, then
    - A → C

Explanation:
This rule shows that dependency can be transferred through an intermediate attribute.

Importance / Uses of Armstrong's Axioms:

- Used to **derive new functional dependencies**.
- Helps in **finding candidate keys**.
- Essential for **normalization** and database design.
- Assists in checking whether a dependency is **logically implied** by others.

(c) Given a relation R(P, Q, R, S, T) and Functional Dependency set FD = { PQ → R, S → T }, determine whether the given R is in 2NF? If not convert it into 2 NF. (7 mark)

Second Normal Form (2NF) is a normalization rule that ensures a relation is in **First Normal Form (1NF)** and that **no non-prime attribute is partially dependent on any candidate key**. The main objective of 2NF is to remove **partial dependency**, which occurs when a non-key attribute depends on **part of a composite primary key** rather than the whole key.

Step 1: Identify Candidate Key of Relation R

Relation: **R(P, Q, R, S, T)**

- From **PQ → R**, attributes **P and Q together determine R**.
- From **S → T**, attribute **S determines T**, but S does not determine all attributes.

To find the candidate key, we consider attributes that are **not functionally dependent on others**:

- P, Q, and S are not determined by any other attributes.

Candidate Key = {P, Q, S}

This set can determine all attributes:

- PQ → R
- S → T

Step 2: Identify Prime and Non-Prime Attributes

- **Prime Attributes:** P, Q, S (part of candidate key)

- **Non-Prime Attributes:** R, T

Step 3: Check for Partial Dependency

A relation violates **2NF** if:

- It has a **composite primary key**, and
- A **non-prime attribute** is dependent on a **part of the key**.

Check Functional Dependencies:

1. **PQ → R**

- R (non-prime) depends on **PQ**, which is a **part of the candidate key (PQS)**
  ✓ This is a **partial dependency**

2. **S → T**

- T (non-prime) depends on **S**, which is also a **part of the candidate key**
  ✓ This is also a **partial dependency**

Conclusion of Analysis:

❌ The given relation **R is NOT in Second Normal Form (2NF)** because it contains **partial dependencies**.

Step 4: Convert the Relation into 2NF

To convert into 2NF, we **remove partial dependencies** by decomposing the relation.

Decomposition:

Relation R1(P, Q, R)

- Functional Dependency: **PQ → R**
- Candidate Key: **PQ**

Relation R2(S, T)

- Functional Dependency: **S → T**
- Candidate Key: **S**

Relation R3(P, Q, S)

- Candidate Key: **PQS**
- Used to maintain the original key relationship

Step 5: Check 2NF for New Relations

- **R1:**
  - Key = PQ
  - R depends on full key → ✓ In 2NF
- **R2:**
  - Key = S
  - T depends on full key → ✓ In 2NF
- **R3:**
  - Contains only key attributes → ✓ In 2NF

Final Answer / Result:

- Original relation **R(P, Q, R, S, T)** is **NOT in 2NF** due to partial dependencies.
- After decomposition, the relations in **2NF** are:

1. **R1(P, Q, R)**

2. **R2(S, T)**
3. **R3(P, Q, S)**

Example (in words):

This situation is similar to a table storing **Order, Product, Supplier, and Supplier Contact** together. Some details depend only on part of the key, causing redundancy. Splitting the table removes redundancy and ensures better structure.

Q.4 (a) Illustrate various storage strategies. (3 mark)

Introduction / Definition:
In a **Database Management System (DBMS)**, **storage strategies** define how data is **physically stored, organized, and accessed** on secondary storage devices such as disks. Efficient storage strategies improve **data retrieval speed, space utilization, and overall database performance**. Different strategies are used based on access patterns and application requirements.

Various Storage Strategies:

1. **Heap (Unordered) File Organization:**

- Records are stored **without any specific order**.
- New records are placed in the **first available free space**.
- Suitable when **frequent insertions** are required.
- Searching a record requires **linear search**, which may be slow.

2. **Sequential (Ordered) File Organization:**

- Records are stored in a **sorted order** based on a key field.
- Provides **fast access** for sequential processing and range queries.

- Insertion and deletion are **costly**, as records may need reorganization.

3. **Hashed File Organization:**

- A **hash function** is applied to a key to determine the storage location.
- Allows **very fast access** to individual records.
- Suffers from **hash collisions**, which require collision-handling techniques.

4. **Indexed File Organization:**

- Uses an **index structure** to quickly locate records.
- Index acts as a **pointer** to actual data blocks.
- Improves search performance but requires **extra storage space** for indexes.

Purpose / Key Points:

- Storage strategies help in **efficient data access**.
- Choice of strategy depends on **type of operations** (search, insert, update).
- Proper storage design reduces **disk I/O operations**

(b) Explain authorization and authentication with respect to database security. (4 mark)

Introduction / Definition:
Database security is concerned with protecting data from **unauthorized access, misuse, and modification**. Two fundamental concepts used to ensure database security are **authentication** and **authorization**. These mechanisms ensure that only **legitimate users** can access the database and perform permitted operations.

Authentication:

Definition:
Authentication is the process of **verifying the identity of a user** before allowing access to the database system. It answers the question: **"Who are you?"**

Explanation / Methods:

- Users are verified using **usernames and passwords**.
- Advanced methods include **biometric authentication**, digital certificates, and **multi-factor authentication**.
- Authentication is performed **before** any database operation is allowed.

Purpose / Advantages:

- Prevents **unauthorized users** from accessing the database.
- Ensures only **valid and registered users** enter the system.

Authorization:

Definition:
Authorization is the process of **granting or denying permissions** to authenticated users for accessing database objects. It answers the question: **"What are you allowed to do?"**

Explanation / Features:

- Controls access to **tables, views, procedures, and data**.
- Permissions include **SELECT, INSERT, UPDATE, DELETE**.
- Managed using commands like GRANT and REVOKE.

Purpose / Advantages:

- Protects sensitive data from **unauthorized operations**.

- Ensures users access only the **data relevant to their role**.

Difference between Authentication and Authorization:

Aspect	Authentication	Authorization
Meaning	Verifies user identity	Assigns access rights
Question answered	Who are you?	What can you do?
Order	Done first	Done after authentication
Example	Login using password	Permission to read a table

(c) Which kind of queries are solved using division operator? Explain in detail. (7 mark)

Introduction / Definition:

In **Relational Algebra**, the **division operator (÷)** is a special and powerful operator used to answer **"for all" type queries**. It is particularly useful when we want to find **entities that are related to all entities in another relation**. The division operator is not a basic operator but is derived from fundamental relational algebra operations.

Purpose of Division Operator:

The division operator is mainly used to solve queries where:

- A condition must be satisfied for **all values** in a given set.
- The query involves **universal quantification** (i.e., "for every").
- We want to find tuples in one relation that are associated with **every tuple in another relation**.

General Form of Division Operator:

Let:

- Relation **R(X, Y)**
- Relation **S(Y)**

Then:

- **R ÷ S = T(X)**

Meaning:

- The result **T** contains those values of **X** that are related to **all values of Y present in S**.

Type of Queries Solved Using Division Operator:

The division operator is used to solve **universal queries**, such as:

- Find students who have **taken all courses** offered by a department.
- Find suppliers who supply **all parts** required by a company.
- Find employees who have **worked on all projects**.
- Find customers who have **purchased all products** of a category.

These queries cannot be easily solved using simple selection or join operations.

Explanation with Conceptual Example (in words):

Consider two relations:

- **Enrollment(Student, Course)**
- **Required_Courses(Course)**

The query:

“Find students who have enrolled in **all required courses**.”

This is a **division-type query** because:

- Student is related to Course in Enrollment.
- We want students who are related to **every course** listed in Required_Courses.

Using division:

- **Enrollment ÷ Required_Courses**
- The result gives **Student IDs** who satisfy the “all courses” condition.

Working of Division Operator (Step-wise Explanation):

1. Identify the **relation containing pairs** (e.g., Student–Course).
2. Identify the **relation containing required values** (e.g., Course).
3. Apply the division operator to check **complete matching**.
4. Output only those entities that are related to **all required tuples**.

The operation internally uses **projection, Cartesian product, and difference** to ensure completeness of matching.

Key Features of Division Operator:

- Used for **“for all” queries**, not “exists” queries.
- Requires **matching attributes** between relations.
- Result contains only **attributes not common** with the divisor relation.
- Often used in **advanced query processing**.

Advantages:

- Simplifies complex universal queries.
- Provides a **clear and mathematical way** to express "all" conditions.
- Useful in **query optimization and conceptual database design**.

Limitations:

- Not directly supported in **SQL syntax**.
- Must be implemented using **joins, grouping, and subqueries**.
- Can be difficult to understand without strong relational algebra knowledge.

Applications:

- Academic databases (students and courses).
- Supply chain and inventory systems.
- Project management databases.
- Requirement compliance queries.

OR

Q.4 (a) Differentiate dynamic hashing and static hashing. (3 mark)

Introduction:
Hashing is a file organization technique used in DBMS to provide **fast access** to records using a **hash function**. Based on how the hash table size is managed, hashing is classified into **static hashing** and **dynamic hashing**. The key difference lies in their ability to **handle data growth and shrinkage** efficiently.

Difference between Static Hashing and Dynamic Hashing:

Basis	Static Hashing	Dynamic Hashing
Hash table size	Fixed at the time of creation	Grows or shrinks dynamically
Number of buckets	Constant	Variable
Handling overflow	Uses overflow chains or buckets	Avoids overflow by bucket splitting
Performance	Degrades as file grows	Maintains good performance
Reorganization	Not supported	Supported automatically
Complexity	Simple to implement	More complex than static hashing

(b) Explain ACID properties of transaction. (4 mark)

A **transaction** in a database is a sequence of operations performed as a single logical unit of work, which must either be fully completed or fully failed. To ensure reliability, consistency, and integrity of data in a database, every transaction must satisfy the **ACID properties**. ACID stands for **Atomicity, Consistency, Isolation, and Durability**. These properties are essential for maintaining database correctness, especially in multi-user and concurrent environments.

1. Atomicity:

- A transaction is atomic, meaning it is **indivisible**.
- Either all operations in the transaction are executed successfully, or none are.

- If any operation fails, the transaction is rolled back, ensuring no partial updates are left in the database.
- Example: Transferring money from one account to another — either both debit and credit happen, or none.

2. Consistency:

- Transactions must transform the database from **one consistent state to another consistent state**.
- All defined rules, constraints, and relationships must be preserved.
- Ensures that after a transaction, the integrity of data is maintained.
- Example: Total amount in accounts should remain consistent after money transfer.

3. Isolation:

- Each transaction should execute **independently** without interference from other concurrent transactions.
- Prevents issues like dirty reads, lost updates, and temporary inconsistencies.
- Helps maintain correct results even when multiple users access the database simultaneously.

4. Durability:

- Once a transaction is committed, its results are **permanently saved** in the database.
- The changes survive system crashes, power failures, or errors.
- Ensures that committed data is never lost.

Importance:

- Maintains **data integrity** and correctness.

- Enables safe **concurrent access** in multi-user databases.
- Ensures reliable **recovery** in case of system failure.

In summary, **ACID properties ensure that database transactions are reliable, consistent, and safe**, making them a fundamental concept in database management systems.

(c) Describe query processing with neat diagram. (7 mark)

Introduction / Definition:
Query processing is the mechanism by which a database management system (DBMS) interprets and executes a query written in a high-level language such as SQL. The main goal is to **retrieve data efficiently and correctly** while optimizing performance. Every query issued by a user goes through multiple steps before the result is returned. Efficient query processing is critical for the **speed, accuracy, and resource management** of a database system.

Detailed Explanation:
Query processing involves **transforming a high-level SQL query into an efficient low-level query plan** that the DBMS can execute. The process ensures that the database retrieves the correct data while using minimal resources like CPU, memory, and I/O operations.

The process can be divided into the following stages:

1. Parsing and Translation:

- The SQL query is first **parsed** to check syntax correctness.
- A **parse tree** is created, representing the logical structure of the query.
- Semantic checks are performed to ensure that tables, columns, and operations exist and are valid.

- Example: SELECT Name FROM Student WHERE Age > 20; → Parse tree shows “SELECT Name” and “condition Age > 20”.

2. Query Optimization:

- The parsed query is converted into **query evaluation plans**.
- The query optimizer selects the most efficient plan using techniques like **join order optimization, selection pushdown, and indexing**.
- Cost-based or heuristic-based optimization ensures **minimum execution cost**.

3. Query Execution:

- The DBMS executes the optimized query plan.
- Data is retrieved using methods like **table scan, index scan, or hash join** depending on the plan.
- Intermediate results are stored in memory or temporary tables as needed.

4. Result Return:

- The final result is formatted and sent back to the user or application.
- Any necessary sorting, grouping, or aggregation is performed before returning results.

Key Features of Query Processing:

- Converts high-level SQL to low-level executable code.
- Optimizes resource usage and execution time.
- Handles **complex queries** with multiple joins, subqueries, and conditions.

- Ensures correctness and consistency of retrieved data.

Advantages:

- Faster query execution through optimization.
- Efficient utilization of CPU and memory.
- Supports large-scale, multi-user database environments.

Disadvantages:

- Query optimization can be computationally expensive.
- Complexity increases for very large or highly complex queries.

Applications:

- Used in all relational DBMS such as MySQL, Oracle, SQL Server.
- Critical for business intelligence, reporting systems, and online transaction processing (OLTP).

Example Explained in Words:
Suppose a user requests all students with marks greater than 80. The SQL query is:
SELECT Name FROM Students WHERE Marks > 80;

- The DBMS parses this query to check syntax.
- The optimizer chooses to use an **index on Marks** to quickly locate qualifying rows.
- Execution retrieves the names of students meeting the condition and returns them to the user.
- In a diagram (if drawn), you would show **User → SQL Query → Parser → Optimizer → Query Plan → Execution Engine → Result Set**.

Q.5 (a) Explain working of two phase locking protocol. (3 mark)

Introduction / Definition:
Two-Phase Locking (2PL) is a concurrency control protocol used in database systems to ensure **serializability** of transactions. It controls how transactions acquire and release locks on database objects, preventing conflicts and maintaining **data consistency** in multi-user environments.

Working of Two-Phase Locking:
The protocol works in **two distinct phases**:

1. Growing Phase:

- In this phase, a transaction **acquires all the locks** it needs (shared or exclusive).
- No locks are released during this phase.
- The transaction can continue requesting additional locks as needed.

2. Shrinking Phase:

- Once the transaction **releases its first lock**, it enters the shrinking phase.
- During this phase, the transaction **cannot acquire any new locks**.
- Locks are gradually released until the transaction completes.

Key Points / Features:

- Ensures **serializability**, meaning transactions appear to execute in some sequential order.
- Prevents **conflicts** like lost updates, dirty reads, and uncommitted data.

- Requires transactions to follow strict lock acquisition and release rules.

Example (in words):

- Transaction T1 wants to read A and write B. It first acquires a **shared lock** on A, then an **exclusive lock** on B (growing phase).
- After performing operations, it releases locks on A and B (shrinking phase).

Advantages:

- Guarantees **correct execution** of concurrent transactions.
- Maintains database **consistency and integrity**.

(b) Explain GRANT, REVOKE and SAVEPOINT commands with suitable example. (4 mark)

Introduction / Definition:
In SQL, **data security and transaction control** are essential. Commands like **GRANT, REVOKE, and SAVEPOINT** help manage **user privileges** and **transaction checkpoints** in a database. These commands ensure that only authorized users can access or modify data and allow partial rollback of transactions when needed.

---

1. GRANT Command

- Used to **give privileges** to users on database objects such as tables, views, or procedures.
- Privileges include **SELECT, INSERT, UPDATE, DELETE**, etc.
- Syntax:
- GRANT privilege_name ON object_name TO user_name;
- **Example:**

- GRANT SELECT, INSERT ON Students TO User1;

This allows User1 to read and insert data into the Students table.

Advantages:

- Enables **controlled access** to database objects.
- Helps **maintain security** by granting only necessary rights.

---

2. REVOKE Command

- Used to **remove previously granted privileges** from users.
- Syntax:
- REVOKE privilege_name ON object_name FROM user_name;
- **Example:**
- REVOKE INSERT ON Students FROM User1;

This removes the ability of User1 to insert data into the Students table.

Importance:

- Prevents unauthorized operations.
- Maintains **database integrity** by controlling user actions.

---

3. SAVEPOINT Command

- Used within a transaction to **set a checkpoint** so that the transaction can be partially rolled back if needed.
- Syntax:
- SAVEPOINT savepoint_name;
- **Example:**
- SAVEPOINT sp1;

- INSERT INTO Students VALUES (101, 'John', 22);
- ROLLBACK TO sp1;

Here, the insertion can be undone without affecting other operations performed before the SAVEPOINT.

Advantages:

- Allows **partial rollback** instead of rolling back the entire transaction.
- Useful in complex transactions with multiple operations.

(c) Assume table CUSTOMER (Cust_Id,Customer_name, Age,Address,Salary). Write a PL/SQL function which givens total number of customers having salary more than one lac per month. (7 mark)

Introduction / Definition:
PL/SQL is Oracle's procedural extension of SQL, used to write **procedures, functions, and triggers** to perform complex database operations. Functions in PL/SQL are **named blocks of code** that accept input parameters, perform operations, and return a single value. In this question, we need a function to **count the total number of customers** whose salary exceeds one lakh per month. This is a common example of using **PL/SQL for data retrieval and processing**.

PL/SQL Function Explanation and Steps:

1. Create the Function

- The function will **accept no input** because we want a total count of all qualifying customers.
- It will **return a number** indicating the total count.

- Inside the function, we will use a SELECT COUNT(*) INTO statement to get the required value.

2. Function Code:

```
CREATE OR REPLACE FUNCTION getHighSalaryCustomers
RETURN NUMBER
IS
   total_count NUMBER;  -- variable to store total customers
BEGIN
   -- Count customers with salary > 100000
   SELECT COUNT(*) INTO total_count
   FROM CUSTOMER
   WHERE Salary > 100000;

   RETURN total_count;  -- return the result
END;
/
```

3. How It Works:

- The IS section declares a variable total_count to store the result.
- The SELECT COUNT(*) INTO total_count queries the CUSTOMER table to find the number of customers with Salary > 100000.
- The RETURN statement returns the count to the caller.

4. Calling the Function:

- You can call this function in SQL or PL/SQL block as:

```
DECLARE
```

```
    high_salary_count NUMBER;
BEGIN
    high_salary_count := getHighSalaryCustomers;
    DBMS_OUTPUT.PUT_LINE('Total Customers with Salary > 1 Lakh: ' || high_salary_count);
END;
/
```

- The output will display the total number of customers earning more than one lakh.

Key Features:

- Encapsulates logic in a reusable function.
- Uses **SELECT INTO** to retrieve data into PL/SQL variable.
- Returns a numeric value usable in other SQL/PLSQL operations.

Advantages:

- Simplifies querying for repeated operations.
- Improves code reusability and readability.
- Ensures **accuracy** by performing server-side calculation.

Applications:

- Financial reports (e.g., counting high-income clients).
- Payroll and HR database operations.
- Any analytical operation needing aggregated data with conditions.

Example Explained in Words:

- If the CUSTOMER table has 100 entries and 15 customers earn more than 1 lakh, calling the function getHighSalaryCustomers will return **15**.
- This can be printed using DBMS_OUTPUT.PUT_LINE or used in another query.

OR

Q.5 (a) Differentiate between conflict and view serializability with respect to transaction. (3 mark)

Introduction / Definition:
In database management, **serializability** ensures that the concurrent execution of transactions produces a **correct and consistent result**, equivalent to some serial execution of the same transactions. There are two main types of serializability: **Conflict Serializability** and **View Serializability**. Both ensure correctness but differ in how they determine if a schedule is serializable.

---

1. Conflict Serializability:

- Based on **conflicting operations** (read/write conflicts) between transactions.
- Two operations conflict if they **access the same data item** and **at least one is a write**.
- A schedule is conflict serializable if it can be **reordered by swapping non-conflicting operations** to form a serial schedule.
- Ensures strict order of operations to avoid inconsistencies.
- Example: If T1 writes X and T2 reads X, the order of these operations matters for conflict serializability.

2. View Serializability:

- Based on the **final outcome (view) of the data** rather than individual conflicts.
- A schedule is view serializable if it produces the **same final values of data items as some serial schedule**, even if some conflicting operations appear out of order.
- More flexible than conflict serializability but harder to check.
- Example: Even if T1 and T2 read and write X in different orders, as long as the final database values match some serial order, the schedule is view serializable.

Key Differences:

Feature	Conflict Serializability	View Serializability
Based on	Conflicting operations (read/write)	Final values (view) of data
Flexibility	Less flexible	More flexible
Checking	Easier (using precedence graph)	Harder to check
Guarantee	Always ensures consistency	Ensures final state consistency

---

(b) Categorize joins in the SQL. Explain each with suitable example. (4 mark)

Introduction / Definition:
In SQL, a **JOIN** is used to combine rows from two or more tables based on a related column between them. Joins are essential for **retrieving meaningful data** from multiple tables in a relational

database. SQL supports different types of joins, each serving a specific purpose depending on how you want to combine data.

Categories of Joins in SQL

1. INNER JOIN:

- Retrieves **only matching rows** from both tables.
- If a row in one table does not have a match in the other table, it is **excluded**.
- **Example:**

```
SELECT Customers.Cust_Id, Customers.Customer_name, Orders.Order_Id

FROM Customers

INNER JOIN Orders

ON Customers.Cust_Id = Orders.Cust_Id;
```

This returns only customers who have placed orders.

---

2. LEFT JOIN (or LEFT OUTER JOIN):

- Retrieves **all rows from the left table** and matching rows from the right table.
- If no match exists, the right table columns return **NULL**.
- **Example:**

```
SELECT Customers.Cust_Id, Customers.Customer_name, Orders.Order_Id

FROM Customers

LEFT JOIN Orders
```

ON Customers.Cust_Id = Orders.Cust_Id;

This returns all customers, including those who have not placed any orders.

---

3. RIGHT JOIN (or RIGHT OUTER JOIN):

- Retrieves **all rows from the right table** and matching rows from the left table.
- If no match exists, the left table columns return **NULL**.
- **Example:**

SELECT Customers.Cust_Id, Customers.Customer_name, Orders.Order_Id

FROM Customers

RIGHT JOIN Orders

ON Customers.Cust_Id = Orders.Cust_Id;

This returns all orders, even if some orders are not linked to a customer in the Customers table.

---

4. FULL OUTER JOIN:

- Retrieves **all rows from both tables**, matching where possible.
- If there is no match, columns from the other table return **NULL**.
- **Example:**

SELECT Customers.Cust_Id, Customers.Customer_name, Orders.Order_Id

FROM Customers

FULL OUTER JOIN Orders

ON Customers.Cust_Id = Orders.Cust_Id;

This returns all customers and all orders, showing NULL where no match exists.

Advantages of Using Joins:

- Combines data from multiple tables efficiently.
- Helps in **complex queries and reports**.
- Avoids data redundancy by normalizing tables.

(c) Write a PL/SQL trigger where employee of "GTU Private Ltd." company cannot update database on 23-Dec-2024 due to maintenance. (7 mark)

Introduction / Definition:
A **PL/SQL trigger** is a stored procedure in Oracle that is automatically executed (or "fired") in response to certain events on a table or view. Triggers are commonly used for **enforcing business rules, auditing, or preventing unauthorized operations**. In this example, we will create a trigger to **prevent employees of "GTU Private Ltd." from updating the database on a specific maintenance day** (23-Dec-2024).

Detailed Explanation and Steps:

1. Requirements:

- Trigger should **fire before UPDATE** on the employee table.
- Check the **current date**; if it matches 23-Dec-2024, the update should be **prevented**.
- Raise an **error message** to notify users about maintenance.

2. Trigger Code:

```
CREATE OR REPLACE TRIGGER prevent_update_maintenance
BEFORE UPDATE ON Employee
FOR EACH ROW
BEGIN
    -- Check if the current date is 23-Dec-2024
    IF TO_DATE('23-DEC-2024','DD-MON-YYYY') =
TRUNC(SYSDATE) THEN
        -- Raise an error to prevent update
        RAISE_APPLICATION_ERROR(-20001, 'Database update is
not allowed today due to maintenance.');
    END IF;
END;
/
```

How the Trigger Works:

1. BEFORE UPDATE ON Employee → Ensures the trigger fires **before any update** is applied.
2. FOR EACH ROW → Applies the trigger to **every row** being updated.
3. TO_DATE('23-DEC-2024','DD-MON-YYYY') = TRUNC(SYSDATE) → Compares current system date with the maintenance date.
4. RAISE_APPLICATION_ERROR → Stops the update and displays a **custom error message**.

Key Features:

- Prevents unauthorized updates on a **specific date**.
- Can be modified for other **maintenance schedules or events**.
- Ensures **data integrity and compliance** during restricted periods.

Advantages:

- Automatically enforces rules **without manual intervention**.
- Avoids accidental data modifications during maintenance.
- Provides clear feedback to users with **custom error messages**.

Applications:

- Database maintenance windows.
- Security enforcement (preventing changes on holidays or restricted days).
- Business rules automation in HR, finance, or inventory systems.

Example Explained in Words:

- Suppose an employee tries to update their salary on 23-Dec-2024:

```
UPDATE Employee
SET Salary = 80000
WHERE Emp_Id = 101;
```

- The trigger checks the date and immediately raises an error: *"Database update is not allowed today due to maintenance."*
- The update **fails**, protecting the database during maintenance.

9ss